

# **BSoD/Introduction Python Scripting**

by Stephen Swaney (stivs)



# Table of Contents

TARGET AUDIENCE.....	5
PRE-REQUISITES.....	5
GOALS.....	5
BPy API Documentation and References.....	5
Text Editor.....	6
The TextWindow Toolbar.....	6
The TextWindow.....	7
The Mouse.....	7
The HotKeys.....	8
Hints.....	8
Hello World.....	9
Comments.....	9
Discussion:.....	10
Objects.....	10
Getting Objects.....	11
Get By Name using Object.Get().....	11
A quick note about 3D Coordinates.....	12
Getting Objects By Selection - Object.getSelected().....	12
Program: PRINT PROPERTIES.....	13
Setting Properties.....	14
Program: MOVE.....	14
Error Handling and Exceptions.....	15
Discussion of Script Links.....	16
Types of Script Links.....	17
The Registry.....	18
Storing and retrieving persistent data in scripts.....	18
Program: Using Persistent Data.....	18
General Math.....	19
A quick discussion of Vector math.....	19
How to create a vector.....	19
A vector example.....	20
Application:Flocking (on preparation).....	20
Discussion of Blender data structures.....	20
Creating Objects.....	21
Objects 101.....	21
Add a Scene.....	22
Add a Lamp.....	22
Add a Camera.....	22
The Whole Scene.....	22
More Objects.....	23
Mesh.....	23
Program:New Mesh.....	24
Creating simple Materials and linking them to Objects.....	25
Program: New Material.....	25
Importers-Exporters.....	26
Python File Input/Output.....	26
Program: Read/Write.....	26
Program: Export.....	27
Program: Import.....	27



## TARGET AUDIENCE

BPy Boot Camp is an introduction to writing Python scripts in Blender. It is targeted at intermediate Blender users. You do not need to know Python to start but you will need to read the appropriate sections in the Python tutorial at [www.python.org](http://www.python.org) . Python is an easy language to learn and there are many on-line resources to help you.

## PRE-REQUISITES

Here are the prerequisites for this course:

- Blender installed
- Use the latest Blender version 2.42a. Other versions will work, but there will be minor differences
- Blender comes with a Python interpreter built in. You do not need a full Python install to begin the course. You can install Python later. The Blender forums are a good resource for help.
- Basic Blender concepts - You need to be familiar with
- Blender Windows and how to change from one type to another
- Adding and Selecting objects
- Running Blender from the command line
- If you are not familiar with the Python programming language, you will want to go thru the tutorial on the Python web site (which will require a full python installation)
- Python Tutorial: · <http://docs.python.org/tut/tut.html>
- other useful python documents for beginners: · <http://www.python.org/doc/>

## GOALS

At the end of this course you will

- be able to create simple scripts to automate repetitive tasks
- understand Blender Data Structures
- start to build a personal code library

## BPy API Documentation and References

The BPy API documentation is an essential reference for writing scripts. It is available browsing on-line and can be downloaded in HTML format for use on your local machine.

<http://www.blender.org/documentation/242PythonDoc/index.html>

The Blender Wiki is an on-line User's Guide and Reference Manual for Blender

<http://mediawiki.blender.org>

Python documentation, references and Python itself are at <http://www.python.org>

# Text Editor

Blender comes with a built-in text editor - the Text Editor window. Originally designed for making notes about a .blend file, it has evolved into a Python programming editor.

The Text Editor uses both menus and hot keys. It will be familiar to anyone who has ever edited a document.

## The TextWindow Toolbar



Text Toolbar

Text Toolbar	
<b>Window type</b>	As with every window header, the first button allows you to set the window type.
<b>Menu</b>	The triangular button expands/collapses menus. Menus provide a self-explanatory way to access all Blender functions which can be performed in the Text Window. They are context sensitive and will change depending on the situation.
<b>Full window</b>	Maximise the window, or return to the previous window display size; return to the previous screen setting ( <b>CTRL+UPARROW</b> )
<b>Line numbers</b>	This button toggles showing of the line numbers on and off.
<b>Syntax Highlighting</b>	This button toggles syntax highlighting. Colors are set in the theme.
<b>Text menu</b>	Choose another Text from the list of available Buffers. The option "Add New" opens a new empty buffer. The option "Open New" turns the Text Window into a File Selection Window and allows you to load a Text Buffer from the disk.
<b>TX:</b>	Give the current Text buffer a new and unique name. After the new name is entered, it appears in the list, sorted alphabetically.
<b>Unlink text</b>	The current Text Buffer is unlinked.
<b>Font size</b>	Allows you to switch from 12 to 15 point size for text.
<b>Tab</b>	Sets the number of spaces a tabs is equal too.

## The TextWindow

```
1 #####
2 #
3 # Demo Script for Blender Manual
4 #
5 #####S68
6 # This script generates polygons, It is quite useless
7 # since you can do polygons with ADD->Mesh->Circle
8 # but it is a nice complete script examle, and the
9 # polygons are 'filled'
10 #####
11
12 #####
13 # Importing modules
14 #####
15
16 import Blender
17 from Blender import NMesh
18 from Blender.BGL import *
19 from Blender.Draw import *
20
21 import math
22 from math import *
23
24 # Polygon Parameters
25 T_NumberOfSides = Create(3)
26 T_Radius        = Create(1.0)
27
28 # Events
29 EVENT_NOEVENT = 1
30 EVENT_DRAW    = 2
31 EVENT_EXIT    = 3
```

The TextWindow is a simple but useful Texteditor, fully integrated into Blender. The main purpose of it is to write Python scripts, but it is also very useful for writing comments in the Blendfile or for instructing other users of the purpose of the scene.

## The Mouse

**LMB** – Sets the cursor position. Also used for selection with **LMB+DRAG**. You can also select with **LMB SHIFT+LMB**

**MMB** – Pan / translates window.

**RMB** – Opens a menu:

- ***New***: Creates a new empty buffer.
- ***Open***: Turns window into a File Selection Window for loading.
- ***Save***: Save text buffer to disk.
- ***Save as***: Turns window into a File Selection Window

## The HotKeys

HotKey	Function	HotKey	Function
<b>ALT+C</b> or <b>CTRL+C</b>	Copy the marked text into a temporary buffer.	<b>ALT+S</b>	Saves the Text buffer.
<b>SHIFT+ALT+F</b>	Opens the same menu as <b>RMB</b>	<b>ALT+U</b> or <b>CTRL+Z</b>	Undo.
<b>ALT+J</b>	Asks for a line number and makes the cursor jump to it.	<b>SHIFT+CTRL+Z</b>	Redo.
<b>ALT+M</b>	Converts all the text in the buffer into a 3D text object (Max 1000 chars.).	<b>ALT+V</b> or <b>CTRL+V</b>	Paste the marked text from the temporary buffer.
<b>ALT+O</b>	Opens a File selecte.	<b>ALT+X</b> or <b>CTRL+X</b>	Cut the marked text into a temporary buffer.
<b>ALT+P</b>	Executes the Text as a Python script.	<b>TAB</b>	Indents the selected block of code.
<b>ALT+R</b>	Reopen current text file.	<b>SHIFT+TAB</b>	Unindents the selected block of code.

**Tip for Windows users:** Blender's temporary buffer is separate from Window's clipboard. To access Window's clipboard use **CTRL+SHIFT+C**, **CTRL+SHIFT+V** and **CTRL+SHIFT+X**.

## Python specific features

Most are under the Format menu.

**Comment / UnComment:** useful for creating blocks of comments or for temporarily disabling blocks of code.. Select the text to be commented (uncommented) and choose from the menu.

**Indent /Unindent:** Python uses indentation to structure code. These commands change the indentation level. Like Comment/Uncomment, select the text and choose from the menu.

## Hints

- Do **NOT** mix spaces and tabs. Use one or the other. The requirement for all bundled scripts is tabs.
- To run a Python script, use either **File->Run Python Script** or press **ALT+P**
- the error message “SyntaxError: invalid syntax” is often caused by improper indentation.



# Hello World

Traditionally, when learning a new programming language, the first program you write is called Hello World. It simply prints out "Hello, World!". Simple, but it demonstrates you have the ability to create and run programs. Python is meant to be an easy-to-use language, so it is not surprising that our Hello World is simple. Here is the whole thing:

```
print 'Hello, World!'
```

The print command writes a text string - the characters in quotes - to the console or terminal window where we started Blender. Quite useful for sending messages and keeping track of what is happening!

- start Blender from a console or terminal window
- change one window to a Text Editor
- choose File-New to get a blank text page
- enter the text below exactly as written:

```
print 'Hello, World!'
```

- run the script by one of the following methods:
  - press **ALT+P** or
  - choose File ->Run Python Script from the drop-down menu or
  - select Execute Script using the Right Mouse menu in the Text window
- our message appears in the console window

Remember that Syntax Error mentioned above?

- just for fun, insert a space before the word 'print' and run the script
- the message 'Python script error, check console' means go look at the terminal or console window where you started Blender. Useful information is there!

## Comments

Comments are simply notes in the code. They do NOT get executed. They are useful for describing what the code does. This is not only helpful to others looking at your script, but also reminds you what you were doing when you look at your code a week later. Good comments describe why something is done. The code itself describes what is being done.

Comments come in a couple different styles:

```
# this is a comment.  comments start with the '#' character
```

```
# the print command prints a string of  
# characters to the console
```

```
print 'Hello, World!' # another silly comment
```

```
# the string of characters is enclosed in quotes  
# either single quote marks 'like this'  
# or double quote marks "like this"
```

```
'''
```

```
This is a block comment.  
The comment starts with three quotes  
- either single or double - on a line  
Use three matching quotes to close the comment  
'''
```

This may seem trivial, but we now know how to

- create a script
- run a script
- print useful messages with the print command

Since that was so easy, let's try something else. Enter the code below and run it.

```
import sys  
print dir(sys)
```

## Discussion:

Our new commands here are the 'import' command and dir(). Python is divided up into modules that contain useful commands and data for specific purposes like talking to the operating system or doing math. Because Python tries not to eat up lots of resources, you must explicitly name modules you want to use using the import command.

The dir() command is used to list the methods and data in a module or that a particular type supports. Unlike the command line interpreter, the Blender interpreter does not print the results of the last command executed, therefore we need to use 'print dir(sys)' to see the contents of the sys module.

Modules are organized as hierarchies. To access a module item, prefix it with the module name followed by a dot like this: print sys.path. Try adding that to your simple script and run it.

All of the Blender BPy API is contained in a module called 'Blender' You will become used to seeing

```
import Blender
```

You can read more about modules and the dir() command here:

<http://www.python.org/doc/current/tut/node8.html>

## Objects

In the world of programming, the word 'object' has a specific meaning - the encapsulation of data and methods. But do not be scared! For our purposes the everyday meaning of the word will do. Objects are things around us - a book, a dog, a computer. They have properties, like color, or mass. They often have behaviors like 'fetch', 'chase-your-tail' or 'run-a-program'. Obviously, not all objects have the same properties or behaviors. However, all objects belonging to the same category or class do have similar properties.

Properties have a name and a value. An object may have the property named 'color'. A value for 'color' is 'green'.

It is important to draw a distinction between a class or type and an instance of that class. There is a type of animal called Dog. My dog is a specific instance of Dog. We can refer to this instance as myDog. One of the properties of myDog is 'name'. The value of 'name' for myDog is 'Spot'. All objects of a particular type have the same properties, but the properties for each instance have their own values.

Blender has many types of objects. Some examples are Cameras, Scenes, Meshes, Empties and IPO Curves. All Blender geometry objects like Cameras and Meshes have some common properties like 'name', 'location', 'size' and 'rotation'.

Blender objects are contained in a module named 'Blender'. Modules must be imported before they can be used. Therefore, one of the first statements in our scripts is

```
import Blender
```

Time for some examples! Start Blender. If your default scene has a Cube, fine. If not, add a Cube. Select it. If you look down in the lower left-hand corner of the 3d view, you see the name of our selected object - Cube, in this case.

Add another Cube and select it. Notice the name in the lower left is now Cube.001. This is Blender's standard naming convention, adding a number after the name. You can also see the name in the Buttons Window if you select the Editing (F9) context - Ob:Cube.001

These names are important because we can locate an object by its name.

All the properties and methods of Blender objects are listed in the API documentation for the Object Module. For now, the only ones we will concern ourselves with are name and location.

## Getting Objects

In order to access the properties of an object, we need a way to refer to a specific object. One way to do this is by the Object name.

### Get By Name using Object.Get()

Create a new, blank Text window and enter the following code. You can leave off the comments (everything on a line after the '#') if you feel lazy.

```
import Blender

ob = Blender.Object.Get('Cube')
print ob           # the object
print ob.name      # object name
print 'location', ob.loc # object location
```

Run this by pressing ALT-PKEY or use 'Execute Script' from the RightMouse menu

Our output looks something like this:

```
[Object "Cube"]
Cube
location (0.0, 0.0, 0.0)
```

What is all this?

[Object "Cube"] - this is simply a character string that represents the object. All Blender objects have a printable representation. It is not always useful. Do not confuse the string representation of an object with the object itself.

Cube - the name of our object, same as in the 3D view and the Editing context

(0.0, 0.0, 0.0) - a set of 3 numbers that represents the location of our object. This sounds complicated so it is time to talk about 3D coordinates.

Notice that we referenced the Get() method by its full hierarchical name, Blender.Object.Get(). Later we will discuss a shorter way to do this.

## A quick note about 3D Coordinates

Blender objects live in a 3 dimensional world. This is equivalent to saying we need a set of three numbers to represent the position or rotation of a Blender object. Our 3 dimensional Blender world is divided up into a grid. The three numbers correspond to positions on the coordinate axes and are named X, Y, and Z. You may remember these from math class as Cartesian Coordinates.

## Getting Objects By Selection - Object.getSelected()

We have talked about getting objects by using their name, but that presents a chicken-and-egg problem. We can get a reference to an object if we know its name and we can get its name if we have a reference to it. One way out of this dilemma is to select an object and then run our script.

Continuing with our current scene with two cubes, select both Cubes and run the following code.

```
import Blender as B

# returns a list of selected objects
ob_list = B.Object.GetSelected()
print ob_list
```

Our output looks something like this:

```
[[Object "Cube"], [Object "Cube.001"]]
```

The method Object.GetSelected() returns a list of selected objects. You can access the first object in a list like this:

```
print ob_list[0]
```

We can access each item in the list in turn by using a 'for' statement like this:

```
for i in ob_list:
    print i
```

When we run it, we get

```
[Object "Cube"]
[Object "Cube.001"]
```

The meaning of the 'for' statement is "for each element in the list, assign reference to the element ( our loop variable 'i' ) and do whatever is in the body of the loop.

Now is a good time to remind you about the python tutorial at [www.python.org](http://www.python.org).

lists: <http://docs.python.org/tut/node5.html#SECTION0051400000000000000000>

for statement: <http://docs.python.org/tut/node6.html#SECTION0062000000000000000000>

The Object module and its Get() and GetSelected() methods:

<http://www.blender.org/documentation/242PythonDoc/Object-module.html>

## **Program: PRINT PROPERTIES**

Let's put together what we know so far and make a script to print out the name and location of all selected objects.

```
import Blender as B

selection = B.Object.GetSelected()

for i in selection:
    # print name and location
    print 'object', i.name, i.loc
```

Our output looks something like this:

```
object Cube.001 (4.6216516494750977, -0.28958892822265625,
-0.0001121363093261607)
object Cube (0.0, 0.0, 0.0)
object Camera (0.0, -7.0, 0.0)
```

Try adding the camera or a lamp to the selection and run the script again.

Notice that our import statement is slightly different here `Import Blender as B`

This form of the import statement creates an alias 'B' for the Blender module. If we do this, later we can refer the elements in the Blender module as B.something rather than by there full name of Blender.something. In this particular case we referenced the GetSelected() method as

```
B.Object.GetSelected()
```

## Setting Properties

Now that we have looked at getting property values, the next obvious question is how to set them. To change the name of an object, we get a reference to it as usual and simply assign a new value of the correct type.

```
import Blender as B

# a list of selected objects
selection = B.Object.GetSelected()
ob = selection[0]    # the first selected object

ob.name = 'MyObj'    # name it 'MyObj'

B.Redraw(-1)         # redraw Blender's Interface
```

Something new here is the line `B.Redraw( -1 )`

This command tells Blender to redisplay the entire interface. When called without an argument, `Redraw()` only redisplay a 3d window. `Redraw()` is an alias for the `Window.Redraw()` command.

### Program: MOVE

We can change other properties in addition to the object name. Let's change the location and size of our object

```
import Blender as B

# a list of selected objects
selection = B.Object.GetSelected()
ob = selection[0]    # the first selected object
ob.LocX += 0.5       # increment the X position by 0.5
B.Redraw( -1 )       # redraw Blender's Interface
```

Our new feature here is

```
ob.LocX += 0.5 # increment the X position by 0.5
```

Previously, we use `ob.loc` to get a set of X, Y and Z coordinates as a tuple or set of numbers. This time, we are accessing the X coordinate as a single number. The `+=` syntax is shorthand for adding the number on the right-hand side to whatever is on the left hand side: `ob.LocX = ob.LocX + 0.5` in this case. This type of expression is common enough to deserve its own abbreviation.

Enter this script into a new window and run it a few times. Notice how the selected object moves across the view.

Try running this script with no object selected to see what happens.

# Error Handling and Exceptions

An important part of writing computer programs - which is what our scripts are - is anticipating and handling what can go wrong. A traditional method of dealing with errors is to return an error status or code after each method or function we call. This looks something like this:

```
error = do_something()
if error:
    # code to handle the error
    # ...

error = do_something_else()
if error:
    # code to handle this error
    # ...
```

Not very pretty, is it? It is also hard to extend when creating functions. Python has a nice feature called 'Exceptions' for dealing with errors. Essentially, you try to do something and if it works, everything is fine. If not, an object called an Exception is created. Normally, you create a block of code to handle the exception. If you don't, the Python interpreter stops and prints the text representation of the exception.

The syntax for exception handling looks like this:

```
try:
    do_something()
    do_something_else()
except:
    # handle any problems here
    # ...
```

A real example will make this clearer. First, let us try getting a non-existent object by name. Using our default scene with a Cube, enter and run the code below in a Text window:

```
import Blender as B

B.Object.Get('non-existent-object')
```

In our output window we see something like this:

```
Traceback (most recent call last):
File "Text.002", line 3, in ?
ValueError: object "non-existent-object" not found
```

When Object.Get() is called with a name like 'non-existent-object', it creates or 'throws' a ValueError exception. Since there is no code to handle the exception object, the interpreter prints it out and stops. Not exactly nice behavior! Here is the syntax we use to process the exception ourselves:

```
try:
    B.Object.Get('non-existent-object')
except ValueError:
    print 'Sorry. Object not found!'
```

The code in the try block is executed. If our object is found, the exception block is ignored. If a ValueError is thrown, then the exception block is executed.

You can read more about Python exceptions here: <http://docs.python.org/tut/node10.html>

Note that you can have multiple except clauses, each with a different exception or, for the last one, a default.

## Discussion of Script Links

In a previous section, we moved an object across the view by repeatedly pressing **ALT+P** to run a script. Experienced Blender users know they can do essentially the same thing automatically with an IPO curve. Since no one wants to repeatedly press keys when we can write scripts, let's try to make this happen automatically.

Scriptlinks come to our rescue. A Scriptlink associates a script with an event and an object. See Module API\_Related doc.

Scriptlinks are created and managed using the Script context in the Buttons Window.

Right now, let's just focus on connecting a script with a Scene and the FrameChanged event. Start with our usual default scene with a cube named 'Cube'. In the Button Window, select the Script context (between the Logic and Shading) Create the following script and name it 'Move'. (You name your script in the dropdown box in the Text Window header in the TX:<text file name>" input area)

```
import Blender as B

NAME = 'Cube'

ob = B.Object.Get( NAME )
ob.LocX += 0.5
```

First:

- Test the script for errors by running it manually.

Then:

- Make sure the Enable Script Links is selected
- press Scene ScriptLink New and choose our script from the drop down menu
- select the FrameChanged event from the event menu

Notice there is no Redraw() in our script. Blender always does a redraw after a frame change.

Test the ScriptLink by pressing the RightArrowKey to advance one frame. Watch our cube move one half a Blender unit in the X direction.

What happens when you activate the animation with **ALT+A**? The cube runs out of the view, advancing once per frame. If we want to run our animation a second time, we have a problem: the cube is already way off to the right. Let's fix it by doing some initialization. Change the 'Move' script as shown in the next page:



```

import Blender as B

NAME = 'Cube'          # object to move
START = 1              # initial frame

ob = B.Object.Get( NAME )
frame = B.Get('curframe')

if frame == START:
    ob.LocX = 0.0
else:
    ob.LocX += 0.5

```

When we run the animation, we see our cube move as before. Now however, when we reset back to frame one, the cube moves back to the origin.

A couple important points about writing scripts:

- Notice we put some constants, NAME and START at the beginning of the script. This makes them easy to find and change. Having magic numbers and names scattered throughout your script is a bad idea.
- We first wrote a simple script to solve a problem. Unfortunately, it was not a complete solution, so we added some new features. This is a common way to build scripts:
  - Solve part of the problem.
  - When that works, grow the script into a complete solution.

## Types of Script Links

Scripts can be connected to Objects as well as Scenes.

Object scriptlinks can trigger on the following events:

- Frame Change
- Render
- Redraw

Scene scriptlinks can trigger on these events:

- OnLoad
- OnSave
- Render
- Redraw
- FrameChange

There is another type of scriptlink called a Space Handler, but we will save discussion of those for another section.

# The Registry

## Storing and retrieving persistent data in scripts

Whenever a Blender script runs, it has no memory of anything that has happened before. When the script exits, nothing is saved. If we want to communicate information between one run of a script or between two different scripts, we need some way to create persistent data. The way we do this is the Registry module.

The Registry module is a persistent dictionary of dictionaries. Said another way, we can create a Python dictionary, stuff it with whatever data we find useful and store it in the Registry by a unique name.

### note:

1. Read about dictionaries in the Python tutorial.
2. Read about the Registry module in the BPy API documentation

In this example, we create a dictionary called myDict with one name/value pair called 'greeting'. Each time the script runs, it changes the greeting from 'hello' to 'goodbye' and prints a message.

This may look complicated, but we can break it down like this:

*First we check if our dictionary is in the registry.*

*if not in the registry, we create it  
otherwise  
print the greeting and swap goodbye for hello  
or vice versa*

You can uncomment the last line and run the script to clear our data from the Registry.

### Program: Using Persistent Data

```
import Blender as B

# is our dictionary in the registry?

myDict= B.Registry.GetKey('Greetings')

if not myDict: # our dictionary does NOT exist. must create
    myDict = {}
    myDict['greeting'] = 'hello'
    print 'creating dictionary!'
else:
    if 'greeting' not in myDict:
        myDict['greeting'] = 'hello'
        print 'no greeting found!'
    else:
        print myDict['greeting'], 'World!'
```

```

# set new greeting. flip/flop between hello & goodbye
if myDict['greeting'] == 'hello':
    myDict['greeting'] = 'goodbye'
else:
    myDict['greeting'] = 'hello'

# store our dictionary
B.Registry.SetKey('Greetings', myDict)

# uncomment this line to remove our dictionary
#B.Registry.RemoveKey('Greetings')

```

## General Math

Python itself has a math module that contains common math functions like sin, cos, and sqrt among others. If you need a basic math operator, it is probably in there. You can read about the math module here <http://docs.python.org/lib/module-math.html>

## A quick discussion of Vector math

The BPy API contains a Mathutils module for doing vector and matrix math. An in-depth discussion of vector and matrix math is beyond our scope here. One quick on-line intro is here: <http://mathforum.org/~klotz/Vectors/vectors.html>

For our purposes now, a Vector is a set of 3 numbers that represents a point or direction in 3D space. Treating the 3 coordinates as a set makes them easier to manipulate. Vectors can be added, subtracted and multiplied using their own set of rules.

A matrix is a set of numbers ( often a 4x4 array ) that can describe the location, position, and scale of an object. Basic math operators can be applied to matrices using their own set of rules.

Both Vector and Matrix are types in the Mathutils module

### How to create a vector

The code snippet below creates a new vector and initializes it to zeros.

```

# import Mathutils and create an alias
from Blender import Mathutils as Math

# create new vector 'v'
v = Math.Vector( 0.0, 0.0, 0.0 )
print v

```

The method Math.Vector() is called a constructor. You can initialize the vector with 3 numbers, a tuple or a list. If you do not provide any arguments, you get a vector with 0.0, 0.0, 0.0. The following snippet is an example of each argument type

```

import Blender as B
from Blender import Mathutils as Math

```

```

print Math.Vector( 0.0, 1.0, 2.0 )      # 3 floats
print Math.Vector( ( 0.0, 1.0, 2.0 ) )  # a tuple
print Math.Vector( [ 0.0, 1.0, 2.0 ] )  # a list
print Math.Vector()                     # no arguments

```

## A vector example

Remember our program to move a cube? Here is how it looks using Vectors

```

import Blender as B
from Blender import Mathutils as Math

ob = B.Object.Get('Cube')

# create vector from loc
loc_vec = Math.Vector(ob.loc)

# create update vector
delta_pos = Math.Vector( 0.5, 0.0, 0.0 )

# add delta_pos to loc_vec
loc_vec += delta_pos

# update object location to new value
ob.loc = loc_vec

B.Redraw()

```

## Application:Flocking

On preparation!!

## Discussion of Blender data structures

Earlier, we talked about Blender objects and their properties. Now that we are smarter, it is time to take a deeper look. Blender objects are actually compound objects. Almost all objects consist of two parts:

- an Object part that holds the spatial information like location, rotation and scale
- an ObData part that contains the geometry for the graphic element. For a Mesh this is the vertices, edges and faces. For a Curve, this is a list of control points.

Now is a good time to look at the Outliner window. The Outliner shows the relationships and linkages between Blender objects.

Create a simple scene with a Cube, a Camera and a Lamp. Change one window to the OOPs window and switch to the Outliner view with View->Show Outliner if the Outliner is not already selected. The Tree structure you see is a representation of all the Blender objects in a Scene. Select View->Hide/Show All to expand the tree.

**Note:** notice how all the objects in the tree are children of a Scene. More on this later.

You can get more information on the Outliner view on the Blender Wiki.

We already know how to get the Object part of a Blender element using the `Object.Get()` or `Object.GetSelected()` methods. How do we access the `ObData` part of a Blender element? Easy! Each Object has an attribute called 'data' that is a reference to the Object's `ObData`.

Here is a simple script to print the names and types of selected objects. Create a simple scene, select the objects to view and run the script. Compare the script output with what you see in the Outliner view.

```
import Blender as B

# get a list of selected objects
objects = B.Object.GetSelected()

# for each object in our list
# print its name and type
# get the ObData part
# print the name and type of the ObData

for i in objects:
    print i.name, type(i)
    obdata = i.data
    print obdata.name, type(obdata)
```

So, in summary:

- Blender has a list of Scenes
- Each Scene has a list of Objects
- Each Object has a link to an `ObData`

## Creating Objects

### Objects 101

We create a basic scene with Camera and Lights.

In order to have a place for our new objects, we need a scene. The Scene does not have an `ObData` but it does have a list of children.

Once we have a scene, we can add objects. They all follow the same pattern.

- create the Object
- create the `ObData`
- link the `ObData` to the Object
- link the Object to a Scene.
- move and rotate the object if necessary

## Add a Scene

First we do our scene like this:

```
scene = B.Scene.New('MyScene')    # a new scene called MyScene
scene.makeCurrent()                # make this the current scene
```

## Add a Lamp

```
# create object, obdata and link
lamp_obj = B.Object.New('Lamp')
lamp_data = B.Lamp.New('Lamp')
lamp_obj.link(lamp_data)
scene.link(lamp_obj)
# position object
lamp_obj.loc = 0, 0, 10            # position at x,y,z
```

That last line 'lamp\_obj.loc =' might seem a little funny since it looks like we are assigning three values to a single attribute. However, Python knows that lamp\_obj.loc is tuple of three numbers so it tries to help you out. We could assign to the LocX, LocY and LocZ attributes individually, but that is more work! Sometimes it is good to be lazy, if you can be smart about it.

## Add a Camera

```
# create object, obdata and link
cam_obj = B.Object.New('Camera')
cam_data = B.Camera.New('ortho')
cam_obj.link(cam_data)
scene.link(cam_obj)
# position object
cam_obj.loc = (0, -15, 2)
# rotate camera to look towards origin
cam_obj.rot = math.radians(90), 0, 0
```

## The Whole Scene

Here is the whole thing together. As an added bonus, at the end, we print the names of all the objects in our new scene:

```
import Blender as B
import math

print '\n***'

# create a new scene
scene = B.Scene.New('MyScene')
scene.makeCurrent()
```

```

print 'current scene is', scene.name

cam_obj = B.Object.New('Camera')
cam_data= B.Camera.New('ortho')
cam_obj.link(cam_data)
scene.link(cam_obj)
scene.setCurrentCamera( cam_obj)
cam_obj.loc = (0, -15, 2)
# rotate camera to look towards origin
cam_obj.rot = math.radians(90), 0, 0

lamp_obj = B.Object.New('Lamp')
lamp_data = B.Lamp.New('Lamp')
lamp_obj.link( lamp_data)
scene.link(lamp_obj)
lamp_obj.loc = 0, 0, 10

B.Redraw(-1) #update the User Interface

# print the names of all objects in our scene
for i in scene.getChildren():
    print i.name

```

## More Objects

Now that we can create a new scene with a camera and a lamp, let's add some more interesting objects.

### Mesh

Blender has two mesh modules - NMesh and Mesh. We are going to use Mesh. This one is newer and more actively supported. Unlike Mesh, the older NMesh module makes copies of any data it uses. This can be slow and use up lots of memory when dealing with large meshes.

**note:** the plural of vertex is vertices.

From editing Meshes as a Blender user, you know a Mesh is made up of vertices, edges and faces. Each vertex is a 3 dimensional point with x,y,z coordinates. Internally, a Mesh holds 3 lists, one for vertices, one for edges and one for faces.

The vertex list is simply all the mesh vertices in no particular order. The other lists reference the vertices by their position in the list, starting with zero.

Since an edge has a vertex at each end, the edge list stores pairs of vertices using the index of each vertex in the vertices list.

Similarly, a face consists of 3 or 4 vertices so we store the face information as triplets or quads of vertices again using the index number from the vertex list.

## Program:New Mesh

This script creates a triangular mesh with a single face

```
import Blender as B

# vertices
p1 = [0,0,0]
p2 = [1,0,0]
p3 = [0.5, 1, 1]

verts = [p1,p2,p3]
faces = [0,1,2]

# create Object
ob = B.Object.New('Mesh', 'MeshOb')

# create ObData
me = B.Mesh.New('myMesh')

# add vertices
me.verts.extend( verts )

# create faces
me.faces.extend( faces )

ob.link( me )
scene = B.Scene.GetCurrent()
scene.link( ob )

B.Redraw(-1)

# print out the verts, edges and faces of our mesh

for v in me.verts:
    print v

for e in me.edges:
    print e

for f in me.faces:
    print f
```



## Creating simple Materials and linking them to Objects

As we have learned, Blender objects consist of two parts, the Object part which holds spatial information and an ObData part that holds the geometry. Blender allows Materials to be linked to either the Object or to the ObData. This choice is set in the User Preferences. For now, we will only consider the case of Materials linked to the ObData.

Blender supports up to 16 materials per object. The items in a material list are numbered from 0 to 15.

For our example, we create 3 materials and apply them to the faces of an existing Cube.

For this example, start with a scene containing a Cube, a Camera and Lamp.

### Program: New Material

```
import Blender as B

# first object in selection list
ob = B.Object.GetSelected()[0]

# get the Mesh version of our cube, not NMesh
me = B.Mesh.Get( ob.data.name )

# create some materials
red = B.Material.New('Red')
red.rgbCol = 1,0,0 # red, green, blue values

green = B.Material.New('Green')
green.rgbCol = 0,1,0 # red, green, blue values

blue = B.Material.New('Blue')
blue.rgbCol = 0,0,1 # red, green, blue values

# add the material list to our ObData
me.materials = [red, green, blue ]

# set the face materials of our cube
# we can set all faces the same color like this
for f in me.faces:
    f.mat = 0

# or for fun, we can alternate colors like this
for i in range(len(me.faces)):
    me.faces[i].mat = i % 3
```

# Importers-Exporters

## Python File Input/Output

Files come in two flavors: text and binary.

- Text files are readable by humans. They contain words or numbers you can read when you print them or look at them in a text editor.
- Binary files contain pure numbers coded as ones and zeros and seem to contain random characters or noise when you look at them. Programs, like Blender, that create lots of numeric data tend to store their data in binary format.

This would be a good time to read Section 7 of the Python Tutorial, particularly 7.2 Reading and Writing Files. If you are reading or writing binary files, the Python struct module is your friend.

### Program: Read/Write

To create a file we use the built-in `open()` command which creates a file object. We need to provide the name of the file and a mode, either read or write. Once we get the file object from `open()` we call `read()` and `write()` methods on it. These methods work much like the print command. When we are done with a file, we call the `close()` method.

Let's modify our old name printing script to write a file of object names, one name per line. This version gets a list of all objects in the current scene, writes their names to a file and then reads them back in, printing to the console.

**note:** the reason we use `scene.getChildren()` rather than `Object.Get()` is that `Object.Get()` returns all the objects in the .blend file. Calling `scene.getChildren()` only returns the objects in the current scene.

```
import Blender as B

scene = B.Scene.getCurrent()
ob_list = scene.getChildren()

# create a file for writing
try:
    outfile = open('myText.txt', 'w')
    for ob in ob_list:
        # write each object name followed by a newline character
        outfile.write( ob.name + '\n' )
    outfile.close()
except:
    print 'Oh no! something went wrong'
else:
    if outfile: outfile.close()
    print 'done writing'
```

```

# open the file we just wrote
try:
    infile = open('myText.txt', 'r')
    for line in infile:
        print line
except:
    print 'Error reading file'
else:
    infile.close()
    print 'done'

```

## Program: Export

Our next example is a simple exporter that writes object names and coordinates to a text file, one object per line. We will use formatted output to create our strings for writing.

```

import Blender as B

scene = B.Scene.getCurrent()
ob_list = scene.getChildren()

# create a file for writing
try:
    outfile = open('export.dat', 'w')
    for ob in ob_list:
        # write each object name followed by a newline character
        outfile.write( "%s %f %f %f\n" % ( ob.name, ob.loc) )
    outfile.close()
except:
    print 'Oh no! something went wrong'
else:
    if outfile: outfile.close()
    print 'done writing'

```

## Program: Import

This example is a simple importer to process data from our exporter. For each line of data, we split the line into a name and the x,y,z coordinates. Then we create an Empty as a ghost in the same location.

### Points to note:

This script has two parts - a main routine and a defined function to handle each line of text we read. By factoring out the `handle_line()` method, we make the complexity of our scripts easier to manage. For a simple script like this, it does not matter as much, but as our scripts get longer and more complex, isolating functionality makes it easier to write and maintain our code. In this example, if we want to change how we handle a line of input, only the `handle_line()` function needs to change. We do not need to touch the main routine.

```

import Blender as B

#
# process each line of a data file
#     create an Empty in the same location as each object
#

def handle_line( line ):
    print line
    # chop up line into name and coordinates
    # remember these are all strings
    name, xs, ys, zs = line.split()
    print name, xs, ys, zs

    scene = B.Scene.getCurrent()
    # create a new Empty named Empty-xxx

    empty = B.Object.New('Empty', 'Empty-' + name)

    # convert our number strings to floats
    empty.loc = float(xs), float(ys), float(zs)
    scene.link(empty)    # link our new object into scene

#
# main
#

infile = open('export.dat', 'r')

try:
    for line in infile:
        handle_line( line )
except Exception, e:
    print 'Oops!', e

B.Redraw(-1)

```